Dive into Golang

@许式伟

2013-07-19

自我介绍

- 七牛
  - 七牛云存储 CEO
  - 《Go语言编程》作者
  - 《Programming in Go》译者
- 盛大
  - 盛大创新院资深研究员
  - 盛大祥云计划（盛大云前身）发起人
  - 盛大网盘发起人
- 百度
  - 百度网页搜索
- 金山
  - 金山软件技术总监
  - WPS Office 2005 首席架构师
  - 金山实验室发起人
    - 研究云存储课题

Golang 思维方式

- 最小心智负担原则
  - 最小特性
  - 最少惊异
  - 最少犯错机会

Golang 思维方式

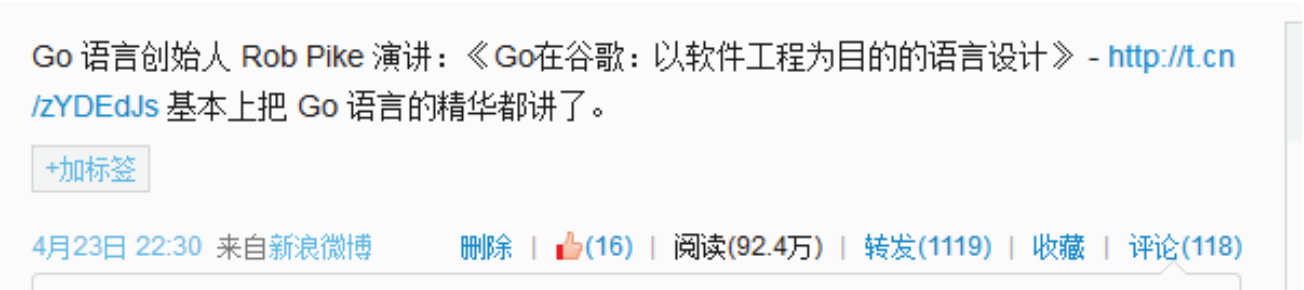
- Go, Next Java? No, Next C!
  - 少就是指数级的多
    - 最少特性原则：如果一个功能不对解决任何问题有显著价值，那么就不提供
  - 显式表达：所写即所得的语言
  - 最对胃口的并行支持
  - 类型系统的纲：interface
  - 极度简化但完备的OOP
    - struct 可以定义成员方法(method)，这是Go对OOP支持的所有内容
    - 简化的符号访问权限控制、显式的 this 指针
  - 错误处理规范
    - 函数多返回值、内置 error 类型、defer
  - 功能内聚：例如，强大的组合能力
  - 消除了堆与栈的边界消除了堆与栈的边界
  - 最友善的 C 语言的支持
  - http://open.qiniudn.com/go-next-c.pptx

Golang 思维方式

- Go, 基于连接与组合的语言
  - Pipeline 与并行模型
    - 在 Go 中实施 Pipeline 非常容易
    - 在 Go 中让任务并行化非常容易
  - 连接
    - Go 组件的连接是松散耦合的。彼此之间有最自然的独立性
    - Go 组件间的协议由 interface 描述，并在编译期进行 check
  - 组合
    - 不支持继承，却胜过继承
    - 不是 COM，但更胜 COM
  - http://open.qiniudn.com/thinking-in-go.mp4

Golang 思维方式

- 以软件工程为目的的语言设计
  - 快速编译
  - 严格的依赖管理
  - 代码风格的强一致性
  - 偏向组合而不是继承
  - http://www.weibo.com/1701886454/ztwNC2uj1



今天讲什么？

- 不讲库
- 不通盘介绍Golang特性
- 打破砂锅问到底
  - 选择Golang局部特性，挖深坑

Dive into Golang

- 切片 (slice)
- 接口 (interface)
- 闭包 (closure)
- 并行编程 (concurrency)

切片 (slice)

- 数据结构
  - Data *type
  - Len int
  - Cap int
- 直观含义：数组片段
  - 指向数组的一个区间(range)
    - [Data, Data+Len)
- 本质：动态数组
  - 可动态扩容的数组(vector)
  - 有 Cap 成员是明证

切片 (slice)

- 取数组片段
  - slice = array[from:to]
  - slice = array[:to]
  - slice = array[from:]
  - slice = array[:]
- reslice
  - slice2 = sliece[from:to]
  - slice2 可以超出 slice 的范围，所以叫 reslice 而不是 subslice
  - slice2 不能超出底层 array 的范围
- 取区间大小/容量 - len(slice) / cap(slice)
- 复制元素 - copy(dest, src)
  - 返回复制的元素个数：min(len(dest), len(src))
  - 实际上如果 dest 是 []byte，那么 src 还可以是 string
- 追加元素
  - append(dest, val)
  - append(dest, v1, v2, ..., vN)
  - append(dest, src...)

切片 (slice)

- 示范代码

```
arr := [6]int{0, 1, 2, 3, 4, 5}
slice := arr[1:3]
slice2 := slice[1:3]
```

- 结果

```
slice.Data = &arr[1]
slice.Len = 3-1 = 2
slice.Cap = 6-1 = 5
slice = {1, 2}
slice2.Data = &slice.Data[1] = &arr[2]
slice2.Len = 3-1 = 2
slice2.Cap = slice.Cap-1 = 4
slice2 = {2, 3}
```

切片 (slice)

- Go 语言为数不多的陷阱之一

```
arr := []int{1, 2, 3, 4, 5}
slice := arr[1:2]
slice = append(slice, 6, 7, 8)
fmt.Println(slice)
fmt.Println(arr)
```

- 信条
  - 不对函数slice类型的参数进行append

接口 (interface)

- 数据结构
  - Data *type
  - Itbl *itbl
- 对比 C++ interface 数据结构
  - vptr *vtable
- 差异
  - Go 的 interface 是个值类型(可定义实例)，里面含有 2 个指针；C++ 的 interface 不能定义实例，只能定义相应的指针类型，比如 IFoo*
- https://github.com/qiniu/gobook/tree/master/chapter9/interface

接口 (interface)

- Go 接口样例

```
var foo IFoo = new(FooImpl)
foo.Bar()
```

  - 翻译成 C

```
FooImpl* unnamed = newFooImpl();
IFoo foo = {unnamed, &FooImpl_IFoo_Itbl};
foo.Itbl->Bar(foo.Data);
```

- C++ 接口样例

```
IFoo* foo = new(FooImpl);
foo->Bar();
```

  - 翻译成 C

```
FooImpl* unnamed = newFooImpl();
IFoo* foo = (IFoo*)unnamed;
foo->vptr->Bar(foo);
```

接口 (interface)

- 赋值 (assignment)
  var foo IFoo = &FooImpl{...}
    - 如果 *FooImpl 类型符合 IFoo 接口
  var foo IFoo = FooImpl{...}
    - 如果 FooImpl 类型符合 IFoo 接口
  var bar IBar = foo
    - 如果 IFoo 接口符合 IBar 接口，也就是 IBar 是 IFoo 要求的子集
  var any interface{} = anyVal
    - 任何类型的实例，都可以赋值给空接口
- 接口查询(query interface)
  w, ok := bar.(io.Writer)
    - 询问 bar 接口指向的组件是否符合 io.Writer 接口
- 类型查询(query type)
  foo, ok := bar.(*FooImpl)
    - 询问 bar 接口指向的组件是否是 *FooImpl 类型
  switch v := bar.(type) { case *FooImpl: ... }
    - 根据 bar 接口指向的组件类型选择

闭包 (closure)

- 原理
  - 闭包只是带有父函数的上下文(Context)的函数
    - 函数带有上下文并不奇怪，函数都可以访问全局变量，那就是上下文。
    - 不同之处在于，父函数本身的状态是动态产生和消亡的，这个上下文需要有生命周期管理。但 Go 是 gc 语言，这一点上也不是问题。
    - 闭包对父函数的Context只是引用而不复制。

闭包 (closure)

- 柯里化 (currying)
  - 对多元函数的某个参数进行绑定

```
func app(in io.Reader, out io.Writer, args []string) { ... }
args := []string{"arg1", "arg2", ...}
app2 := func(in io.Reader, out io.Writer) {
    app(in, out, args)
}
```

  - Go1.1 支持了对 receiver 的快速绑定

```
func (recvr *App) main(in io.Reader, out io.Writer) { ... }
app := &App{...}
app2 := app.main
```

    - 等价于

```
app2 := func(in io.Reader, out io.Writer) { app.main(in, out) }
```

闭包 (closure)

- 闭包的组合

```
func pipe(
    app1 func(io.Reader, io.Writer),
    app2 func(io.Reader, io.Writer)
) func(io.Reader, io.Writer) {

    return func(in io.Reader, out io.Writer) {
        pr, pw := io.Pipe()
        defer pw.Close()
        go func() {
            defer pr.Close()
            app2(pr, out)
        }()
        app1(in, pw)
    }
}
```

闭包 (closure)

- 闭包的陷阱

```
var closures [2]func()
for i := 0; i < 2; i++ {
    closures[i] = func() {
      fmt.Println(i)
    }
}
closures[0]()
closures[1]()
```

  - 不要认为会打印 0 和 1，实际打印是 2 和 2
  - 修正方法

```
var closures [2]func()
for i := 0; i < 2; i++ {
    val := i
    closures[i] = func() {
      fmt.Println(val)
    }
}
closures[0]()
closures[1]()
```

并行编程 (concurrency)

- goroutine
  - 轻量级执行体 (类比：协程/纤程)
  - 无上限 (只受限于内存)、创建/切换成本低
    - 但注意不要当做是零成本，还是应该留心创建 goroutine 频度与数量
- channel
  - 本质上是一个 MessageQueue
  - 非常正统的执行体间通讯设施
- sync.Mutex/RWMutex/Cond/etc
  - 不要把 channel 当做万金油，该 Mutex 还是要 Mutex

并行编程 (concurrency)

- 误区
  - 用 channel 来做互斥 (正常应该让 Mutex 做)
    - 比如多个 goroutine 访问一组共享变量
- channel 的成本
  - 作为消息队列，channel 成本原高于 Mutex
  - 成本在哪？
    - channel 内部有 Mutex，因为它本身属于共享变量
    - channel 内部可能有 Cond，用来等待或唤醒满足条件的 goroutine
    - 出让 cpu 并且让另一个 goroutine 获得执行机会，这个切换周期不低，远高于 Mutex 检查竞争状态的成本(后者通常只是一个原子操作)

并行编程 (concurrency)

- 用 channel 等别人的结果

```
done := make(chan Result, 1)
go func() {
    ...
    done <- Result{}
}()
...
result := <-done
fmt.Println(result)
```

并行编程 (concurrency)

- 不永久等别人的结果，对方可能有异常 (考虑timeout)

```
done := make(chan Result, 1)
go func() {
    ...
    done <- Result{}
}()
...
select {
case result := <-done:
    fmt.Println(result)
case <- time.After(3 * time.Second):
    fmt.Println("timeout")
}
```

并行编程 (concurrency)

- 给多个人同样的活，谁先干完要谁的结果(考虑timeout)

```
done := make(chan Result, 3)
for i := 0; i < 3; i++ {
    go func() {
        ...
        done <- Result{}
    }()
}
...
select {
case result := <-done:
    fmt.Println(result)
case <- time.After(3 * time.Second):
    fmt.Println("timeout")
}
```

并行编程 (concurrency)

- 生产者/消费者模型
  - 并行编程中，多个 goroutine 间符合生产者/消费者模型非常常见。
  - channel 用于生产者和消费者间的通信，并适配两者的速度。
    - 如果生产者速度过快，那么它会 channel 缓冲区满时停下来等待；如果消费者速度过快，则在 channel 缓冲区空时停下来等待。

Q & A

许式伟
@七牛云存储